La respuesta anterior en español es la siguiente:

La función `rec1(n)` es una función recursiva que devuelve el valor de `1` si `n` es menor o igual que `1`, de lo contrario devuelve el producto de dos llamadas recursivas con `n-1` como argumento. Para calcular la complejidad temporal `t(n)` de este algoritmo, necesitamos analizar cuántas veces se llama a la función y cuánto tiempo tarda cada llamada.

Supongamos que cada llamada a la función toma un tiempo constante `c`, independientemente del valor de `n`. Entonces, podemos escribir una relación de recurrencia para `t(n)` de la siguiente forma:

$$t(n) = \begin{cases} c & \text{si } n \leq 1 \\ 2t(n-1) + c & \text{si } n > 1 \end{cases}$$

Esto significa que para cualquier `n` mayor que `1`, la complejidad temporal es el doble de la complejidad temporal de `n-1` más un tiempo constante. Para resolver esta relación de recurrencia, podemos usar el método de iteración. Empezamos sustituyendo `n-1` en la ecuación y simplificando:

$$\begin{aligned} t(n) &= 2t(n-1) + c \\ &= 2(2t(n-2) + c) + c \\ &= 4t(n-2) + 3c \\ &= 4(2t(n-3) + c) + 3c \\ &= 8t(n-3) + 7c \\ &\ldots \end{aligned}$$

Podemos ver que se forma un patrón: el coeficiente de `t(n-k)` es 2^k, y el término constante es $2^k - 1$. Podemos generalizar esto de la siguiente manera:

$$t(n) = 2^k t(n-k) + (2^k - 1)c$$

Para encontrar el valor de `k` que hace que `n-k` sea igual a `1`, podemos resolver para `k` en función de `n`:

$$
\begin{aligned}
n - k &= 1 \\
k &= n - 1
\end{aligned}
$$

Sustituyendo este valor de `k` en la ecuación general, obtenemos:

$$
\begin{aligned}
t(n) &= 2^{n-1}t(1) + (2^{n-1} - 1)c \\
&= 2^{n-1}c + (2^{n-1} - 1)c \\
&= (2^{n} - 1)c
\end{aligned}
$$

Por lo tanto, la complejidad temporal del algoritmo es $O(2^n)$, lo que significa que crece exponencialmente con respecto al tamaño de la entrada. Este es un algoritmo muy ineficiente y se debe evitar si es posible.